

S P E C I F I C A T I O N

IBM Docket No. SVL9-2003-0037-US1

TO ALL WHOM IT MAY CONCERN:

BE IT KNOWN that we, Allen K. Hawley, of San Jose, California and citizen of the United States, and Pavan K. Immaneni of San Jose, California and citizen of India, have invented new and useful improvements in

USER CONFIGURABLE LANGUAGE INDEPENDENT CODE ASSIST ENGINE METHOD, SYSTEM, ARTICLE OF MANUFACTURE, AND COMPUTER PROGRAM PRODUCT

of which the following is a specification:

1
2
3 **USER CONFIGURABLE LANGUAGE INDEPENDENT CODE ASSIST ENGINE**
4 **METHOD, SYSTEM, ARTICLE OF MANUFACTURE, AND COMPUTER PROGRAM**
5 **PRODUCT**
6
7
8
9

10 **CROSS-REFERENCE TO RELATED APPLICATIONS**
11

12 Application Serial Number _____, filed concurrently herewith on December 9, 2003
13 for USER CONFIGURABLE LANGUAGE INDEPENDENT CODE ASSIST METHOD,
14 SYSTEM, ARTICLE OF MANUFACTURE, AND COMPUTER PROGRAM PRODUCT (IBM
15 Docket STL9-2003-0130-US1), currently co-pending, and assigned to the same assignee as the
16 present invention; and

17 Application Serial Number _____, filed concurrently herewith on December 9, 2003
18 for LANGUAGE INDEPENDENT CODE ASSIST USER PREFERENCES CONFIGURATION
19 METHOD, SYSTEM, ARTICLE OF MANUFACTURE, AND COMPUTER PROGRAM
20 PRODUCT (IBM Docket STL9-2003-0131-US1), currently co-pending, and assigned to the same
21 assignee as the present invention.

22 The foregoing co-pending applications are incorporated herein by reference.
23
24
25
26

27 A portion of the Disclosure of this patent document contains material which is subject to
28 copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone
29 of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office
30 patent file or records, but otherwise reserves all copyright rights whatsoever.

1

2 **BACKGROUND OF THE INVENTION**

3

4

5 **1. Field of the Invention**

6

7 [0001] The present invention relates in general to computer programs, and more particularly to
8 providing assistance to a programmer creating computer program code.

9

10

11 **2. Description of the Related Art**

12

13 [0002] A code assist or content assist is a function which aids a programmer in writing a computer
14 program statement by suggesting proposals or candidates for insertion into the computer program
15 statement. The programmer may select one of the proposals or candidates in order to supplement,
16 modify, complete, or replace a computer program statement being edited.

17

18 [0003] Conventional code assist functions are exemplified by the Eclipse Project code assist
19 function as described in the Eclipse Project Java Development User Guide, (c) Copyright IBM Corp.
20 2000, 2001, pages 9-12, <http://www.eclipse.org/eclipse/index.html>. The Eclipse Project code assist
21 function provides a programmer with suggested assistance in writing Java code. The programmer
22 may activate the Eclipse Project code assist function from a Java code editor on a line of Java
23 computer program code which causes a scrollable list of available Java code completions to be
24 displayed. The programmer may then select one of the proposed Java code completion candidates
25 from the scrollable list to be inserted to complete the line of Java computer program code.

26

27 [0004] The programmer may also configure the behavior of the Eclipse Project code assist by
28 selecting or specifying various preferences in a Code Assist tab within a Java Editor preference page.
29 Color preferences allow the programmer to specify the colors used for the code assist window and
30 the colors used when showing method parameters. An alphabetical sort preference, if enabled,
31 causes proposals to be sorted in alphabetical order. A visibility preference limits displayed proposals

1 in the list by Java visibility rules to only those proposals visible in a current invocation context. An
2 argument name preference, if enabled, causes a selection of a Java method proposal to additionally
3 insert the Java method's argument names as specified in the Java method's declaration. A qualified
4 versus import name preference allows the programmer to select whether the code assist function
5 inserts fully qualified names for type proposals or inserts non-qualified names for type proposals in
6 conjunction with an addition of the corresponding import declaration. A preference for automatic
7 insertion of a single proposal when the proposal list only contains the single proposal may be
8 specified. The user may also specify preferences as to the code assist function's activation behavior.
9 An auto activation preference may specify that code assist is invoked automatically in response to
10 specified character triggers after a specified time delay.

11

12 [0005] Even with these capabilities and user-specified preferences, conventional code assist
13 functions still fail to provide adequate assistance for a programming language which allows a
14 variation in valid syntax for a particular command or verb, a variation in keyword representations for
15 a particular keyword, variations in valid argument types for a particular command or verb, or
16 variations in style such as character case. In view of these possible variations, a user may prefer only
17 one of the variations, or the user may prefer to be reminded of all of the possible variations.

18

19 [0006] In particular, these conventional code assist functions fail to provide adequate assistance for
20 variable-syntax programming languages such as COBOL or PL/I having these complexities.
21 Complex languages such as COBOL may have numerous variations in syntax for a single statement.
22 For example, a COBOL GOTO statement has the syntax: "GO" ["TO"] procedure-name, which
23 gives rise to two format variations: "GO TO procedure-name" and "GO procedure-name". The
24 COBOL MOVE statement has the following syntax:

25 "MOVE" [CORRESPONDING | CORR] {identifier-1 | literal-1} "TO" identifier-2

26 and has six format variations:

27 "MOVE CORRESPONDING identifier-1 TO identifier-2"

28 "MOVE CORRESPONDING literal-1 TO identifier-2"

29 "MOVE CORR identifier-1 TO identifier-2"

1 "MOVE CORR literal-1 TO identifier-2"

2 "MOVE identifier-1 TO identifier-2"

3 "MOVE literal-1 TO identifier-2"

4 In a complex language such as COBOL, keywords may have alternative spellings. For example,
5 COBOL keywords may have the following alternative spellings: "CORRESPONDING" or "CORR",
6 "IDENTIFICATION" or "ID", "JUSTIFIED" or JUST", "PICTURE" or "PIC", "SYNCHRONIZED"
7 or "SYNC", and "THROUGH" or "THRU". Alternatively, keywords or reserve words may be
8 optional, such as the optional keyword "TO" in the above GOTO statement or the optional keyword
9 CORRESPONDING in the above MOVE statement. Further complexities may result from
10 keywords dictating a subsequent format of a statement, or from a format being recursive.

11

12 [0007] A programmer desiring an effective code assist function for such a complex programming
13 language needs a code assist function which supports such complexity variations, supports user-
14 specified preferences for such complexity variations, and which effectively provides suggested
15 proposals for completing programming statements having such complexity variations. However,
16 conventional code assist functions fail to provide this. Thus, there is a clearly felt need for an
17 improved code assist function for such complex language variations.

SUMMARY OF THE INVENTION

[0008] Preferred embodiments of the present invention comprise a method, system, article of manufacture, and computer program product for providing code assist.

[0009] In accordance with a preferred embodiment of the present invention, a code assist function which suggests candidates responsive to a parsing of a partial program instruction statement is provided by parsing a program containing the partial program instruction statement to identify variables contained in the program; parsing the partial program instruction statement to generate proposals; and providing proposals to append to the partial program instruction statement to a user responsive to both the parsing of the program and the parsing of the partial program instruction statement. The code assist function may be performed according to a user-selected programming language dependent file selected from a plurality of programming language dependent files.

[0010] A preferred embodiment of the present invention has the advantage of providing an improved code assist function.

[0011] A preferred embodiment of the present invention has the advantage of providing an improved user interface for a code assist function.

[0012] A preferred embodiment of the present invention has the advantage of providing a user-configurable set of user preferences for configuring an operation of an improved code assist function.

[0013] A preferred embodiment of the present invention has the advantage of providing an improved code assist function supporting multiple computer programming languages.

[0014] A preferred embodiment of the present invention has the advantage of providing an

1 improved code assist function supporting multiple syntax and format variations for a single computer
2 programming statement.

3

4 [0015] A preferred embodiment of the present invention has the advantage of providing an
5 improved code assist function supporting multiple keyword variations for a particular keyword of a
6 computer programming statement.

7

8 [0016] A preferred embodiment of the present invention has the advantage of providing an
9 improved code assist function supporting optional keywords for a particular computer programming
10 language command or verb.

11

12 [0017] A preferred embodiment of the present invention has the advantage of providing an
13 improved code assist function supporting computer programming language commands or verbs
14 containing recursive syntax.

15

16 [0018] A preferred embodiment of the present invention has the advantage of providing an
17 improved code assist function providing a list of commands or verbs, key words, identifier names,
18 and literal names as proposals at a current cursor position in a computer program statement.

19

20 [0019] A preferred embodiment of the present invention has the advantage of providing an
21 improved code assist function providing a user-selectable choice of either a shorter list or a longer
22 list of proposals.

23

24 [0020] A preferred embodiment of the present invention has the advantage of providing an
25 improved code assist function providing a user-selectable preference as to a case of a proposal: upper
26 case, lower case, or mixed case.

27

28 [0021] A preferred embodiment of the present invention has the advantage of providing an
29 improved code assist function providing a user-selectable choice from available formats for a
30 particular command or verb.

1 [0022] A preferred embodiment of the present invention has the advantage of providing an
2 improved code assist function providing a user-selectable choice of either full or abbreviated
3 representations of a keyword.

4

5 [0023] A preferred embodiment of the present invention has the advantage of providing an
6 improved code assist function displaying a suggested candidate responsive to a determination of
7 declared variables and constants.

8

9 [0024] A preferred embodiment of the present invention has the advantage of providing an
10 improved code assist function displaying a suggested candidate responsive to a determination of a
11 code segment containing a partial program instruction statement.

BRIEF DESCRIPTION OF THE DRAWINGS

[0025] For a more complete understanding of the present invention and the advantages thereof, reference is now made to the Description of the Preferred Embodiment in conjunction with the attached Drawings, in which:

Figure 1 through Figure 7 are graphical user interfaces used in a preferred embodiment of the present invention;

Figure 8 is a block diagram the preferred embodiment of the present invention;

Figure 9 is a class diagram of the preferred embodiment of the present invention;

Figure 10 through Figure 12 are flowcharts of method steps preferred in carrying out the preferred embodiment of the present invention; and

Figure 13 is a block diagram of a computer system used in performing a method of a preferred embodiment of the present invention, forming part of an apparatus of a preferred embodiment of the present invention, storing a data structure of a preferred embodiment of the present invention, and which may use an article of manufacture comprising a computer-readable storage medium having a computer program embodied in said medium which may cause the computer system to practice a preferred embodiment of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

[0026] An embodiment of the invention is now described with reference to the figures where like reference numbers indicate identical or functionally similar elements. Also in the figures, the left most digit of each reference number corresponds to the figure in which the reference number is first used. While specific configurations and arrangements are discussed, it should be understood that this is done for illustrative purposes only. A person skilled in the relevant art will recognize that other configurations and arrangements can be used without departing from the spirit and scope of the invention. It will be apparent to a person skilled in the relevant art that this invention can also be employed in a variety of other devices and applications.

[0027] Fig. 1 illustrates an editor screen or window 110 displaying a COBOL source code file in which a blank line 120 has been inserted by the user. The user may activate the code assist function by pressing a key or key combination, such as CONTROL-SPACE, which causes the code assist function to display a window 130 containing a list 140 of proposals or proposed candidates to insert at the current cursor position 120. The user may select one of the proposals from the list by clicking a mouse cursor on the proposal to be selected, and responsive to the user's selection of the selected proposal, the selected proposal is inserted at the current cursor position. The user may also scroll the list up and down to find the desired proposal. Alternatively, the user may use keystrokes, such as up arrow, down arrow, tab, return, or enter keys, to scroll, navigate, and select within the list 140.

[0028] If the user desires to insert a MOVE statement at the current cursor position 120, then the user may scroll down the list 140 until a MOVE keyword proposal is displayed. The user may then select the MOVE keyword proposal to be inserted which results in the window 210 of Fig. 2 in which the MOVE keyword 220 is inserted in the partial program statement 230, and in which the current cursor position 240 is moved after the inserted keyword. Responsive to the partial program statement 230 now comprising the MOVE keyword 220, the code assist function suggests further proposal candidates 250 for insertion at the now current cursor position 240. Based upon the syntax of a MOVE program statement, the code assist function suggests proposal candidates comprising the keywords CORRESPONDING and CORR which may follow the MOVE keyword, and comprising identifiers or variables declared within the program which may follow the MOVE keyword, but

1 excluding those identifiers which cannot be specified in a MOVE statement. For example, in some
2 COBOL dialects, data items described with the following types of usage cannot be specified in a
3 MOVE statement: INDEX, POINTER, FUNCTION-POINTER, PROCEDURE-POINTER, and
4 OBJECT REFERENCE.

5

6 [0029] If the user selects the CORR keyword proposal 260 for the next insertion at the current
7 cursor position 240, the window 310 of Fig. 3 results in which the CORR keyword 330 has been
8 inserted at the end of the partial program statement 320 and in which the current cursor position 340
9 is moved after the inserted keyword 330. Responsive to the partial program statement 320 now
10 comprising the MOVE and CORR keywords, the code assist function suggests further proposal
11 candidates 350 for insertion at the now current cursor position 340. Based upon the syntax of the
12 MOVE program statement, the code assist function suggests proposal candidates comprising
13 identifiers or variables declared within the program which may follow the MOVE statement CORR
14 keyword.

15

16 [0030] The user may now select one of the literal or variable proposals from 350 such as the In-
17 Name proposal 360 for the next insertion at the current cursor position 340. The window 410 of Fig.
18 4 results in which the In-Name literal and TO keyword 430 have been inserted at the end of the
19 partial program statement 420 and in which the current cursor position 440 is moved after the
20 inserted literal and keyword 430. Responsive to the partial program statement 420 now comprising
21 the MOVE and CORR keywords followed by the literal In-Name and the TO keyword, the code
22 assist function suggests further proposal candidates 450 for insertion at the now current cursor
23 position 440. Based upon the syntax of the MOVE program statement, the code assist function
24 suggests proposal candidates comprising identifiers or variables declared within the program which
25 may follow the MOVE statement TO keyword.

26

27 [0031] The literal or variable proposal Out-Name 460 from the window 450 may be selected by the
28 user for the next insertion at the current cursor position 440. The window 510 of Fig. 5 results in
29 which the Out-Name literal has been inserted at the end of the program statement 520. This insertion
30 results in a complete program statement 520, and responsive to this complete program statement 520
31 comprising a complete MOVE program statement, the code assist function does not suggest any

1 further proposal candidates 350 for this complete program statement 520.
2

3 [0032] Referring now to Figures 6 and 7, graphical user interfaces which may be used to configure
4 the operation of the code assist function of the preferred embodiment are illustrated. Referring first
5 to Fig. 6, window 605 provides a graphical user interface which enables the user to specify user-
6 selectable preferences as to the operation of the code assist function of the preferred embodiment.
7 The user may select checklist item 610 to cause the code assist function to display a longer list of
8 candidates responsive to the parsing of a partial program instruction statement, or alternatively, the
9 user may select checklist item 615 to cause the code assist function to display a shorter list of
10 candidates responsive to the parsing of the partial program instruction statement. For example, the
11 longer list of candidates may comprise all candidates suggested by the code assist function; whereas,
12 the shorter list may comprise categories of candidates such as keywords, identifiers, or literals from
13 which the user may select to display the candidates in the selected category.

14

15 [0033] As a default behavior of the code assist function, the user may specify a preference as to
16 longer representations displaying full keywords or shorter representations displaying abbreviated
17 keywords in proposal candidates suggested by the code assist function. The user may select checklist
18 item 620 to cause a display of a longer representation of a program instruction keyword candidate
19 responsive to the parsing of a partial program instruction statement, or the user may select checklist
20 item 625 to cause a display of a shorter representation of a program instruction keyword candidate
21 responsive to the parsing of the partial program instruction statement. For example, if the user has
22 selected checklist item 620 indicating a preference for longer full keyword representations, then the
23 code assist function will display in suggested proposal candidates full keywords such as
24 "CORRESPONDING", "IDENTIFICATION", "JUSTIFIED", "PICTURE", "SYNCHRONIZED",
25 and "THROUGH". If the user has selected checklist item 625 indicating a preference for shorter
26 abbreviated keyword representations, then the code assist function will display in suggested proposal
27 candidates shorter abbreviated keywords such as "CORR", "ID", "JUST", "PIC", "SYNC", and
28 "THRU".

29

30 [0034] The user may also specify a preference as to the case of the suggest proposal candidates.
31 The user may select checklist item 630 to cause proposal candidates to be displayed in a lower-case

1 representation (i.e., "move"), checklist item 635 to cause proposal candidates to be displayed in an
2 upper-case representation (i.e., "MOVE"), or checklist item 640 to cause proposal candidates to be
3 displayed in a mixed-case or leading upper-case representation (i.e., "Move").

4

5 [0035] As a further default behavior of the code assist function, the user may specify a preference
6 as to displaying a full syntax representation or a partial syntax representation. The user may select
7 checklist item 645 to cause a display of a full syntax representation of a program instruction verb
8 candidate responsive to the parsing of a partial program instruction statement, or the user may select
9 checklist item 650 to cause a display of a partial syntax representation of a program instruction verb
10 candidate responsive to the parsing of a partial program instruction statement. For example, if the
11 user has selected checklist item 645 indicating a preference for a full syntax representation, then the
12 code assist function will display a MOVE instruction candidate in the full syntax: "MOVE
13 CORRESPONDING IDENTIFIER-1 TO IDENTIFIER-2". If the user has selected checklist item
14 650 indicating a preference for a partial syntax representation, then the code assist function may
15 display a MOVE instruction candidate in a partial syntax: "MOVE IDENTIFIER-1 TO
16 IDENTIFIER-2".

17

18 [0036] Although the user may specify user-selected preferences applicable to all candidates
19 proposed by the code assist function, the user may also override these general preferences with verb-
20 specific preferences by selecting checklist item 655 which causes a display of a list of verbs from
21 which the user may select one verb to specify verb-specific preferences. If the user selects one of the
22 verbs, such as "MOVE", to specify verb-specific preferences, then a window 700 of Fig. 7 is
23 displayed. Even though the user may have specified shorter abbreviated keywords as the default
24 behavior of the code assist function, the user may specify a verb-specific preference of full keywords
25 for the selected verb which overrides the general default behavior. For example, if the user has
26 selected checklist item 705 indicating a preference for longer full keyword representations for the
27 MOVE instruction, then the code assist function will display in suggested proposal candidates full
28 keywords such as "MOVE CORRESPONDING" as opposed to the general preference of shorter
29 abbreviated keywords such as "ADD CORR" or "SUBTRACT CORR". Similarly, checklist item
30 710 allows the user to override a general preference for longer full keywords with a verb-specific
31 preference of shorter abbreviated keywords.

1 [0037] In a similar fashion, the user may use upper-case checklist item 715, lower-case checklist
2 item 720, and mixed-case checklist item 725 to specify a verb-specific preference for the selected
3 verb which overrides the general default behavior specified in window 605.

4

5 [0038] The Statement Syntax section 730 of the verb-specific preferences window 700 allows the
6 user to select one preferred syntax from a plurality of valid syntax representations. For example, for
7 the MOVE instruction, Statement Syntax section 730 allows the user to select from a full keyword
8 full syntax representation 735 comprising “MOVE CORRESPONDING IDENTIFIER1 TO
9 IDENTIFIER2”; an abbreviated full syntax representation 740 comprising “MOVE CORR
10 IDENTIFIER1 TO IDENTIFIER2”, a partial syntax 750 comprising “MOVE IDENTIFIER1 TO
11 IDENTIFIER2”, or an alternative partial syntax 755 “MOVE LITERAL1 TO IDENTIFIER2”. The
12 preferred syntax selected by the user will be used by the code assist function when it suggests
13 proposal candidates for that particular verb.

14

15 [0039] Alternatively, to select the syntax preference for the instruction, the user may click on the
16 checklist item Selectable Statement Syntax 760 which allows the user to click on portions of the
17 syntax diagram to specify the syntax preference. For example, the user may click on “MOVE” 765,
18 “CORRESPONDING” 770, “IDENTIFIER1” 780, “TO” 790, and “IDENTIFIER2” 795 to identify
19 and select the following full keyword full syntax representation of the MOVE instruction: “MOVE
20 CORRESPONDING IDENTIFIER1 TO IDENTIFIER2”. In lieu of selecting
21 “CORRESPONDING” 770, the user may select “CORR” 775 which results in the user identifying
22 and selecting the following abbreviated keyword full syntax representation: ““MOVE CORR
23 IDENTIFIER1 TO IDENTIFIER2”. If the user clicks on “MOVE” 765, “IDENTIFIER1” 780, “TO”
24 790, and “IDENTIFIER2” 795, then the partial syntax “MOVE IDENTIFIER1 TO IDENTIFIER2” is
25 identified and selected as the syntax representation of the MOVE instruction. Alternatively, the user
26 may click on “MOVE” 765, “LITERAL1” 785, “TO” 790, and “IDENTIFIER2” 795 to identify and
27 select the following alternative partial syntax representation of the MOVE instruction: “MOVE
28 CORRESPONDING LITERAL1 TO IDENTIFIER2”.

29

30 [0040] In lieu of these graphical user interface techniques, the user may select the checklist item

1 Edit Syntax Statement File 797 to directly edit the syntax statement rules contained in the syntax
2 statement file used by the code assist function parser.

3

4 [0041] Referring now to **Figure 8**, a block diagram of the preferred embodiment of the present
5 invention is illustrated. A programmer may open a program file 810 with editor 820 to edit program
6 source code contained in the program file 810. If the programmer requests code assist or content
7 assist, it is provided by Code Assist Engine 830 which uses a Syntax Library appropriate for the
8 programming language in which the program of program file 810 is written. If program file 810 is
9 written in COBOL, then the Code Assist Engine 830 uses a COBOL Syntax Library 840 to provide
10 code assist. Alternatively, if the program file 810 is written in another programming language, such
11 as PL/I, C++, or Java for example, then the Code Assist Engine 830 uses either a PL/I Syntax Library
12 850, C++ Syntax Library 860, or Java Syntax Library 870 to provide code assist. These
13 programming languages are merely exemplary examples as a Syntax Library may be written to
14 support any programming language. The Code Assist Engine 830 may determine which Syntax
15 Library is appropriate to use with program file 810 by analyzing either attributes or content of
16 program file 810. For example, if a file attribute such as a file name extension is “cbl” indicating
17 that program file 810 is a COBOL program file, then the Code Assist Engine 830 uses COBOL
18 Syntax Library 840. Alternatively, Code Assist Engine 830 may analyze the content of program file
19 810 to determine that it contains COBOL program code and that COBOL Syntax Library 840 should
20 be used. In still other embodiments, the Syntax Library used by the Code Assist Engine 830 may be
21 based upon a user-specified preference, a user-specified selection, or a history file.

22

23 [0042] Referring now to **Figure 9**, a class diagram of the preferred embodiment of the present
24 invention is illustrated. The classes comprise a CompletionProcessor class 902, Editor class 910,
25 VariablesParser class 908, UserStatement class 906, SyntaxLibrary class 904, CursorEngines class
26 920, Normal CursorEngine class 924, and Recursive CursorEngine class 926.

27

28 [0043] The CompletionProcessor class 902 manages the overall operation of the code assist
29 function, and comprises methods getCurrentText() 914, getImages() 942,
30 CompleteCompletionProposals() 952, and completionProposals() 912. The method getCurrentText()

1 **914** calls the editor **910** for the `elementText()` **916** to get the partial statement to be parsed to provide
2 code assist. The methods `CompleteCompletionProposals()` **952**, `completionProposals()` **912** and
3 `getImages()` **942** match a type from each proposal of the combined proposal vector to an image and
4 return these images to the `CompletionProcessor` **902**.

5

6 **[0044]** The Editor class **910** provides editing functions for the program file being edited, and
7 comprises `elementText()` **916** which provides the partial statement to `getCurrentText()` **914**.

8

9 **[0045]** The VariablesParser class **908** parses the program to identify variables and functional
10 constants, and comprises methods `getVariables()` **938**, `getDivision()` **954**, and
11 `getFigurativeConstants()` **940**. The method `getVariables()` **938** parses the program to identify
12 variables. The method `getDivision()` **954** gets Syntax Statements from SyntaxLibrary **904** to parse a
13 current division of the program file. The method `getFigurativeConstants()` **940** parses the program
14 to identify and get functional constants.

15

16 **[0046]** The UserStatement class **906** parses the partial statement based upon Syntax Statements
17 obtained from the Syntax Library **904**, and comprises methods `parseUserStatement()` **918**, and
18 `getStatement()` **948**. The method `parseUserStatement()` **918** parses the partial statement, and the
19 method `getStatement()` **948** obtains the appropriate Syntax Statements from the Syntax Library **904**.

20

21 **[0047]** The SyntaxLibrary class **904** stores an array of Syntax Statements describing the syntax of a
22 particular programming language, and comprises methods `getVerbs()` **956**, `getKeyWords()` **958**,
23 `isCommand()` **960**, `isKeyWord()` **962**, `getSyntaxStatmnts()` **964**, `getVariables()` **966**, and
24 `findSyntaxStatement()` **968**. The method `getVerbs()` **956** returns the commands supported by the
25 programming language for which the SyntaxLibrary stores the programming language syntax. A
26 command is normally the first keyword of a programming language statement. The method
27 `getKeyWords()` **958** returns the keywords, excluding command keywords, supported by the
28 programming language for which the SyntaxLibrary stores the programming language syntax. The
29 method `isCommand()` **960** tests whether a token is a command. The method `isKeyWord()` **962** tests
30 whether a token is a keyword. The method `getSyntaxStatmnts()` **964** gets the syntax statements as

1 defined in a SyntaxLibrary for a particular programming language. The method getVariables() 966
2 gets the variables defined in the source program as detected by VariablesParser. The method
3 findSyntaxStatement() 968 finds syntax statements in the SyntaxLibrary which match a given
4 command.

5

6 [0048] Within the SyntaxLibrary class 904, language syntax diagrams are converted into syntax
7 statement strings and stored in an array of strings forming a Syntax Statement Library. The first
8 keyword in each syntax statement is the command or verb, and is distinguished from other
9 keywords. When an incomplete programming language statement is presented to the Code Assist
10 Engine 830, it matches the statement command or verb to the command keywords in the syntax
11 statements producing a result comprising those matching syntax statements to be considered for
12 subsequent processing which generates proposals.

13

14 [0049] Within syntax statements, special characters are used to delimit the syntax statements as
15 follows:

<u>Character</u>	<u>Significance</u>
&n, &rn	“&n” and “&rn” are recursion labels. “&n” labels a start of a recursion within a syntax statement wherein n is a number from 0 to 9. An “&n” is paired with an “&rn” which indicates the end of a recursion. The second “&rn” causes a branch back or recursion to the first “&n”.
[]	Brackets indicate Or phrases in which a syntax statement requires either the first item located before the vertical bar within the brackets or the second item located after the vertical bar within the brackets.
{ }	“{” and “}” indicate phrase boundaries wherein once started a phrase must be completed, as opposed to Or phrases which do not have to be completed.
\$	“\$” indicates a conditional which may be applied to keywords (i.e., \$TO), Or phrases (i.e., \$[]), and phrases (i.e., \${ }).
&a	“&a” are symbol labels wherein “a” may be either “i”, “s”, “b”, “l”, “p”, etc. representing identifiers, statements, both, literal, procedure names, etc.

1 [0050] As stated above, a Syntax Statement Library is an array of syntax statement strings using
2 the above notation. Given a syntax diagram for each format of each statement, the syntax statements
3 may be written by matching the syntax diagram to the syntax statement notation, including recursion
4 phrases. Different Syntax Statement Libraries may represent different programming languages
5 without modifying the Code Assist Engine. Furthermore, different Syntax Statement Libraries may
6 be used to represent the same programming language, such as having a different Syntax Statement
7 Library for each COBOL division, like IDENTIFICATION, PROCEDURE, etc.

8

9 [0051] For example, the syntax of a COBOL ALTER statement may be represented by and stored
10 in the Syntax Statement Library by the syntax statement:

11 "ALTER &1 &p TO \${PROCEED TO} &p &r1"

12 Within this ALTER syntax statement, "ALTER", "TO" and "PROCEED" are keywords. "ALTER"
13 as the first keyword in the syntax statement is considered to be the command or verb. Commands
14 are determined by getting the first unique keyword in each syntax statement. "&1" and "&r1" are a
15 recursive pair indicating the recursion "&1 &p TO \${PROCEED TO} &p &r1". "{" and "}" mark a
16 phrase "PROCEED TO" that once started must be completed. The "\$" marks the phrase
17 "{\$PROCEED TO}" as conditional or optional. "&p" represents a procedure symbol. Note that
18 symbols from the syntax statement match identifiers or variables from the user statement and not
19 keywords; whereas, keywords in the syntax statement match keywords in the user statement.

20

21 [0052] The processing of the syntax statements of the Syntax Statement Library occurs in two
22 primary stages: moving the cursor, and creating proposals. The first stage of moving the cursor
23 starts with a character string returned from getCurrentText(). This character string is the partial user
24 statement. The partial user statement is parsed into tokens which are compared one by one against
25 tokens in one or more syntax statements, such as the above mentioned COBOL syntax statement
26 representing the COBOL ALTER command. Tokens are divided into keywords and variables. As
27 long as matching between the user statement tokens and the syntax statement tokens occurs, the
28 cursor is moved to the start of the next token. The cursor moving stage performs the token matching
29 according to the keywords, phrases, and symbols specified in a syntax statement. When all of the
30 tokens from the partial user statement are processed, the cursor moving stage is complete. If
31 necessary, a same portion of a syntax statement (e.g., between "&1" and "&r1") is parsed again to

1 continue matching if a recursion is matched.

2

3 [0053] The second stage of creating proposals starts with a cursor index left by the first stage of
4 moving the cursor, wherein this cursor index indicates a position of the last matching tokens. The
5 second stage then moves forward from this cursor index position to generate one or more proposals.
6 For example, if the first cursor moving stage had matched the partial user statement against the
7 following ALTER syntax statement through the conditional “\$”:

8 "ALTER &1 &p TO \${PROCEED TO} &p &r1"

9 then the second stage of creating proposals starts with a cursor index pointing to the conditional “\$”
10 in the ALTER syntax statement as the first stage has already matched the tokens “ALTER &1 &p
11 TO” between the syntax statement and the partial user statement. The second stage of creating
12 proposals will then generate “PROCEED TO” as a proposal. Since “PROCEED TO” is an optional
13 conditional, the second stage will also generate a proposal without “PROCEED TO” which would be
14 the next token comprising a procedure name.

15

16 [0054] If the cursor from the first stage is indexing the end of a recursion phrase, such as the “&r1”
17 in the above ALTER syntax statement, then the second stage would generate another proposal
18 comprising a second “&p TO &p”, i.e., “procedure-name1 TO procedure-name2”. If the cursor from
19 the first stage is indexing the start of an Or phrase, all of the alternative values within the Or phrase
20 would be generated as proposals.

21

22 [0055] The CursorEngines class 920 creates and controls a CursorEngine for each Syntax
23 Statement, and comprises methods createEngines() 922, generateProposals() 950, and
24 executeCursorEngines() 928. The method createEngines() 922 inspects a Syntax Statement and
25 creates an instance of a normal CursorEngine, and if it detects a recursion loop in the Syntax
26 Statement, then it also creates one instance of a recursive CursorEngine for each recursion loop
27 detected in the statement. The method generateProposals() 950 requests that each Normal
28 CursorEngine() 924 and each recursive CursorEngine() 926 created by its CursorEngines 920 to
29 create proposals based upon their prior parsing of the partial statement. The method
30 executeCursorEngines() 928 requests that each Normal CursorEngine() 924 and each recursive
31 CursorEngine() 926 created by its CursorEngines 920 begin cursor movement and parsing of the

1 partial statement.

2

3 [0056] The Normal CursorEngine class 924 parses a single Syntax Statement, and comprises
4 methods moveSyntaxCursor() 930, parseSyntaxStatement() 944, and createProposal() 934. The
5 methods moveSyntaxCursor() 930 and parseSyntaxStatement() 944 request that a normal
6 CursorEngine parses its Syntax Statement as driven by the parsing of the partial statement. The
7 method createProposal() 934 requests that a Normal CursorEngine() 924 create proposals based upon
8 its prior parsing of the partial statement.

9

10 [0057] The Recursive CursorEngine class 926 parses a single recursion within a single Syntax
11 Statement, and comprises methods moveSyntaxCursor() 932, parseSyntaxStatement() 946, and
12 createProposal() 936. The methods moveSyntaxCursor() 932 and parseSyntaxStatement() 946
13 request that a Recursive CursorEngine parses its Syntax Statement as driven by the parsing of the
14 partial statement. The method createProposal() 936 requests that a Recursive CursorEngine() 924
15 create proposals based upon its prior parsing of the partial statement.

16

17 [0058] Referring now to **Figure 10** through **Figure 12**, the flowcharts 1000, 1100, and 1200
18 illustrate the operations preferred in carrying out the preferred embodiment of the present invention.
19 In the flowcharts, the graphical conventions of a diamond for a test or decision and a rectangle for a
20 process or function are used. These conventions are well understood by those skilled in the art, and
21 the flowcharts are sufficient to enable one of ordinary skill to write code in any suitable computer
22 programming language.

23

24 [0059] Referring first to **Figure 10**, after the start 1010 of the process 1000, process block 1020
25 creates an instance of a CompletionProcessor 902 registered for an appropriate programming
26 language (i.e., cbl file extensions for COBOL), an instance of UserStatement 906, an instance of a
27 VariablesParser 908, and an instance of a SyntaxLibrary 904 for each Division within the program
28 file being edited by editor 910. Process block 1030 monitors for a user requesting the code assist
29 function through the use of a key combination such as Control-Space after entering a partial COBOL
30 statement. Alternatively, the user may activate the code assist function through other input devices
31 and user interfaces well known to those skilled in the art. Responsive to the user's activation of the

1 code assist function, process block **1040** of the CompletionProcessor **902** calls
2 CompletionProcessor#completionProposals() **912**. Process block **1050** of the CompletionProcessor
3 calls getCurrentText() **914** which calls elementText() **916** of the editor **910** to get the partial
4 statement upon which the code assist function was activated. The instance of
5 UserStatement#parseStatement() **918** is called by process block **1060** of CompletionProcessor **902** to
6 begin the proposal generation. UserStatement#parseStatement() **918** parses the partial statement
7 provided by the editor **910** and uses a command (or verb) contained in the partial statement to locate
8 Syntax Statement(s) via getStatement **948**, locating one Syntax Statement for each format variation
9 in the appropriate SyntaxLibrary **904** for the current COBOL Division. Process block **1070** creates
10 one instance of CursorEngines **920** for each Syntax Statement format variation. Control then passes
11 to process block **1120** on **Figure 11**, illustrated by flowchart connectors **A**, **1080** on **Figure 10** and
12 **1110** on **Figure 11**.

13

14 [0060] Referring now to **Figure 11**, after control passes to process block **1120**, process block **1120**
15 of createEngines() **922** inspects its Syntax Statement and creates one normal CursorEngine **924** and
16 one recursive CursorEngine **926** for each recursion loop detected in the statement, if any. Process
17 block **1130** of CursorEngines **920** invokes executeCursorEngines **928** to begin a cursor movement
18 phase. During this cursor movement phase, process block **1140** of the normal CursorEngine **924**
19 asks moveSyntaxCursor() **930** to parse its syntax statement via parseSyntaxStatement() **944** as driven
20 by a parsing of the user-entered partial statement. If a recursive Syntax Statement is detected, then
21 process block **1150** of the normal CursorEngine **924** calls an appropriate recursive CursorEngine **926**
22 which asks moveSyntaxCursor() **932** to parse its recursive Syntax Statement via
23 parseSyntaxStatement() **946** as driven by a parsing of the user-entered partial statement. Cursor
24 movement within the user-entered partial statement continues until a recursive CursorEngine() **926**
25 mis-matches tokens from the user-entered partial statement, or until the recursive CursorEngine()
26 reaches a match between its Syntax Statement and the partial user statement, at which point the
27 recursive CursorEngine **926** returns to the normal CursorEngine **924**. The normal CursorEngine **924**
28 continues to move its cursor until a mis-match occurs, until more recursion syntax is reached, or until
29 the end of the user-entered partial statement is reached. Control then passes to process block **1220**
30 on **Figure 12**, illustrated by flowchart connectors **B**, **1160** on **Figure 11** and **1210** on **Figure 12**.

1 [0061] Referring next to **Figure 12**, after control passes to process block **1220**, process block **1220**
2 begins the proposal generation phase in which `generateProposals()` **950** of `CursorEngines()` **920** asks
3 `Normal CursorEngine()` **924** and `recursive CursorEngine()` **926** to create proposals based upon their
4 prior parsing via `createProposal()` (**934** and **936**). To create these proposals, each `CursorEngine` (**924**
5 or **926**), starting from the current cursor location, moves the cursor forward generating zero or more
6 proposals. Keywords from the Syntax Statement may be added to a proposals vector as proposals. If
7 the current cursor position within the user-entered partial statement is setting on an identifier, then
8 corresponding variables and functional constants are obtained from `VariablesParser#getVariables()`
9 **938** and `VariablesParser#getFigurativeConstants()` **940**. These corresponding variables and
10 functional constants may also be added to the proposals vector. Thereafter, process block **1230**
11 concatenates the proposals (strings and types) vectors from each `CursorEngine` (**924** or **926**) in each
12 instance of `CursorEngines` (**924** or **926**) across an array of format variation(s) to create a single
13 combined proposal vector which is returned to `CompletionProcessor` **902**. Process block **1240** of
14 `CompletionProcessor` **902** causes `completionProposals()` **912** to match a type from each proposal of
15 the combined proposal vector to an image via `getImages()` **942** and to return these images to the
16 `CompletionProcessor` **902** which displays a pop-up window (**130**, **250**, **350**, or **450**) containing the
17 proposal images from which the user can select a proposed Keyword, identifier or constant to
18 continue the entry of the partial statement. After all processing is completed, the process ends at
19 process block **1250**.

20

21 [0062] With reference now to the figures, and in particular with reference to **Figure 13**, there is
22 depicted a pictorial representation of a computer system **1300** which may be utilized to implement a
23 method, system, article of manufacture, data structure, and computer program product of preferred
24 embodiments of the present invention. The block diagram of **Figure 13** illustrates a computer system
25 **1300** used in performing the method of the present invention, forming part of the apparatus of the
26 present invention, and which may use the article of manufacture comprising a computer-readable
27 storage medium having a computer program embodied in said medium which may cause the
28 computer system to practice the present invention. The computer system **1300** includes a processor
29 **1302**, which includes a central processing unit (CPU) **1304**, and a memory **1306**. Additional
30 memory, in the form of a hard disk file storage **1308** and a computer-readable storage device **1310**, is

1 connected to the processor 1302. Computer-readable storage device 1310 receives a computer-
2 readable storage medium 1312 having a computer program embodied in said medium which may
3 cause the computer system to implement the present invention in the computer system 1300. The
4 computer system 1300 includes user interface hardware, including a mouse 1314 and a keyboard
5 1316 for allowing user input to the processor 1302 and a display 1318 for presenting visual data to
6 the user. The computer system may also include a printer 1320.

7

8 [0063] Using the foregoing specification, the invention may be implemented using standard
9 programming and/or engineering techniques using computer programming software, firmware,
10 hardware or any combination or sub-combination thereof. Any such resulting program(s), having
11 computer readable program code means, may be embodied within one or more computer usable
12 media such as fixed (hard) drives, disk, diskettes, optical disks, magnetic tape, semiconductor
13 memories such as Read-Only Memory (ROM), Programmable Read-Only Memory (PROM), etc., or
14 any memory or transmitting device, thereby making a computer program product, i.e., an article of
15 manufacture, according to the invention. The article of manufacture containing the computer
16 programming code may be made and/or used by executing the code directly or indirectly from one
17 medium, by copying the code from one medium to another medium, or by transmitting the code over
18 a network. An apparatus for making, using, or selling the invention may be one or more processing
19 systems including, but not limited to, central processing unit (CPU), memory, storage devices,
20 communication links, communication devices, servers, input/output (I/O) devices, or any sub-
21 components or individual parts of one or more processing systems, including software, firmware,
22 hardware or any combination or sub-combination thereof, which embody the invention as set forth in
23 the claims. User input may be received from the keyboard, mouse, pen, voice, touch screen, or any
24 other means by which a human can input data to a computer, including through other programs such
25 as application programs, databases, data sets, or files.

26

27 [0064] One skilled in the art of computer science will easily be able to combine the software
28 created as described with appropriate general purpose or special purpose computer hardware to
29 create a computer system and/or computer sub-components embodying the invention and to create a
30 computer system and/or computer sub-components for carrying out the method of the invention.
31 Although the present invention has been particularly shown and described with reference to a

1 preferred embodiment, it should be apparent that modifications and adaptations to that embodiment
2 may occur to one skilled in the art without departing from the spirit or scope of the present invention
3 as set forth in the following claims.